Back to Basics: The Special Member Functions

KLAUS IGLBERGER
C++ Trainer/Consultant

Author of the blaze C++ math library

(Co-)Organizer of the Munich C++ user group

Chair of the CppCon B2B and SD tracks

Email: klaus.iglberger@gmx.de

Klaus Iglberger
class Widget
{
  public:
    Widget(); // Default constructor
    Widget( Widget const& ); // Copy constructor
    Widget& operator=( Widget const& ); // Copy assignment operator
    Widget( Widget&& ) noexcept; // Move constructor
    Widget& operator=( Widget&& ) noexcept; // Move assignment operator
    ~Widget(); // Destructor
};
The Special Member Functions

Interactive Task: Name all special member functions (SMF)!

class Widget
{
    public:
    Widget();               // Default constructor
    Widget( Widget const& ); // Copy constructor
    Widget& operator=( Widget const& );  // Copy assignment operator
    Widget( Widget&& ) noexcept; // Move constructor
    Widget& operator=( Widget&& ) noexcept; // Move assignment operator
    ~Widget();                // Destructor
};
The Special Member Functions

Interactive Task: Name all special member functions (SMF)!

```cpp
class Widget {
    public:
        Widget();
        Widget( Widget const& );
        Widget& operator=( Widget const& );
        Widget( Widget&& ) noexcept;
        Widget& operator=( Widget&& ) noexcept;
        ~Widget();
};
```

The Rule of 6
class Widget
{
    public:
        Widget();
        Widget( Widget const& );
        Widget& operator=( Widget const& );
        Widget( Widget&& ) noexcept;
        Widget& operator=( Widget&& ) noexcept;
        ~Widget();
};

Interactive Task: Name all special member functions (SMF)!

The Rule of 5
class Widget
{
    public:
        Widget();
        Widget( Widget const& );
        Widget& operator=( Widget const& );
        Widget( Widget&& ) noexcept;
        Widget& operator=( Widget&& ) noexcept;
        ~Widget();
};
The Special Member Functions

Interactive Task: Name all special member functions (SMF)!

class Widget
{
    public:
    Widget();
    Widget( Widget const& );
    Widget& operator=( Widget const& );
    Widget( Widget&& ) noexcept;
    Widget& operator=( Widget&& ) noexcept;
    ~Widget();
};
The Special Member Functions

Interactive Task: Name all special member functions (SMF)!

class Widget
{
    public:
    Widget(); // Default constructor
    Widget( Widget const& ); // Copy constructor
    Widget& operator=( Widget const& ); // Copy assignment operator
    Widget( Widget&& ) noexcept; // Move constructor
    Widget& operator=( Widget&& ) noexcept; // Move assignment operator
    ~Widget(); // Destructor
};
The Default Constructor
The Default Constructor

The compiler generates a default constructor ...

```cpp
// Compiler-generated default constructor available
class Widget
{
public:
    // ...
};

Widget w1;    // Compiler generated, ok
Widget w2{};  // Compiler generated, ok
```
The compiler generates a default constructor ...
• if no constructor is explicitly declared and ...
• if all data members and base classes can be default constructed.

```cpp
// No compiler-generated default constructor available
class Widget
{
    public:
        Widget( Widget const& ); // <- explicit declaration of the
        // ...                   //    copy ctor -> no default ctor
    //    available

    Widget w1; // No default constructor, compilation failure
    Widget w2{}; // No default constructor, compilation failure
```
The Default Constructor

The compiler generates a default constructor …
• if no constructor is explicitly declared and …
• if all data members and base classes can be default constructed.

```cpp
// No compiler-generated default constructor available
class Widget
{
    public:
        // ...
    private:
        NoDefaultCtor member_; // Data member without default ctor
};

Widget w1; // No default constructor, compilation failure
Widget w2{}; // No default constructor, compilation failure
```
Data Member Initialization

**Interactive Task:** What is the initial value of the three data members \(i\), \(s\), and \(pi\)?

```cpp
struct Widget {
    int i;             // Uninitialized
    std::string s;    // Default (i.e. empty string)
    int* pi;          // Uninitialized
};

int main() {
    Widget w;          // Default initialization
}
```
The compiler generated default constructor ...
• initializes all data members of class (user-defined) type ...
• but not the data members of fundamental type.

```c++
struct Widget
{
    int i;          // Uninitialized
    std::string s;  // Default (i.e. empty string)
    int* pi;        // Uninitialized
};

int main()
{
    Widget w;        // Default initialization: Calls
                        // the default constructor
}
Interactive Task: What is the initial value of the three data members i, s, and pi?

```cpp
struct Widget {
    int i;          // Initialized to 0
    std::string s;  // Default (i.e. empty string)
    int* pi;        // Initialized to nullptr
};

int main() {
    Widget w{};    // Value initialization
}
```
Data Member Initialization

If no default constructor is declared, value initialization ...

• zero-initializes the object

• and then default-initializes all non-trivial data members.

```c
struct Widget {
    int i;           // Initialized to 0
    std::string s;   // Default (i.e. empty string)
    int* pi;         // Initialized to nullptr
};

int main() {
    Widget w{};      // Value initialization: No default
}               // ctor -> zero+default init
```
Data Member Initialization

**Guideline:** Prefer to create default objects by means of an empty set of braces (value initialization).
Data Member Initialization

Interactive Task: What is the initial value of the three data members i, s, and pi?

```
struct Widget
{
  Widget() {}  // Explicit default constructor
  int i;  // Uninitialized
  std::string s; // Default (i.e. empty string)
  int* pi;  // Uninitialized
};

int main()
{
  Widget w{};  // Value initialization
}
```
Data Member Initialization

An empty default constructor ...
• initializes all data members of class (user-defined) type ...
• but not the data members of fundamental type.

```cpp
struct Widget
{
    Widget() {} // Explicit default constructor
    int i;      // Uninitialized
    std::string s; // Default (i.e. empty string)
    int* pi;    // Uninitialized
};

int main()
{
    Widget w{};   // Value initialization: Declared
    // default ctor -> calls ctor
} 
```
Data Member Initialization

= default lets the compiler generate the default constructor.
• = default counts as definition;
• = default may give you a couple of bonus effects (e.g. noexcept).

```cpp
struct Widget
{
    Widget() = default;
    int i;       // Initialized to 0
    std::string s; // Default (i.e. empty string)
    int* pi;     // Initialized to nullptr
};

int main()
{
    Widget w{ };    // Value initialization: Declared
    // default ctor -> calls ctor
}```
Data Member Initialization

**Guideline:** Avoid writing an empty default constructor. Prefer to let the compiler provide a definition or define by `=default`. 
Data Member Initialization

Via the default constructor, we can properly initialize all data members:

```cpp
struct Widget
{
    Widget()
    {
        i = 42; // Initialize the int to 42
        s = "CppCon"; // Initialize the string to "CppCon"
        pi = nullptr; // Initialize the pointer to nullptr
    }

    int i;
    std::string s;
    int* pi;
};
```
Data Member Initialization

Via the default constructor, we can properly initialize all data members:

```cpp
struct Widget
{
    Widget()
    {
        i = 42;       // Assignment, not initialization
        s = "CppCon"; // Assignment, not initialization
        pi = nullptr; // Assignment, not initialization
    }

    int i;
    std::string s;
    int* pi;
};
```
Data Member Initialization

Via the default constructor, we can properly initialize all data members:

```cpp
struct Widget
{
    Widget() :
        s{"CppCon"} // Initialization of the string
        // in the member initializer list
    {
        i = 42; // Assignment, not initialization
        pi = nullptr; // Assignment, not initialization
    }

    int i;
    std::string s;
    int* pi;
};
```
Data Member Initialization

Via the default constructor, we can properly initialize all data members:

```cpp
struct Widget
{
    Widget()
        : i {42}, // Initializing to 42
          s {"CppCon"}, // Initializing to "CppCon"
          pi{}       // Initializing to nullptr
    {}

    int i;
    std::string s;
    int* pi;
};
```
Data Member Initialization

**Guideline:** Remember the responsibilities of the default constructor.

**Core Guideline C.47:** Define and initialise member variables in the order of member declaration

**Core Guideline C.49:** Prefer initialization to assignment in constructors.
The Destructor

The compiler generates the destructor ...

// Compiler-generated destructor available
class Widget
{
    public:
        // ...

};

Widget w1;  // Compiler generated, ok
Widget w2{}; // Compiler generated, ok
The Compiler Generates the Destructor...

• if the destructor is not explicitly declared.

```cpp
// No compiler-generated destructor available
class Widget
{
public:
    ~Widget(); // <- explicit declaration of the destructor ->
    // ... // compiler doesn't generate the destructor
};

Widget w1; // Manual destructor, ok
Widget w2{}; // Manual destructor, ok
```
The compiler generated destructor...
• calls the destructor of all data members of class type;
• doesn’t do anything special for fundamental types.

class Widget
{
    public:
        // ...

        ~Widget()
        {

        }

    // ...
    private: // The three data members:
        int i; // - i as a representative of a fundamental type
        std::string s; // - s as a representative of a class (user-defined) type
        Resource* pr{}; // - pr as representative of a possible resource
};
The compiler generated destructor ...
• calls the destructor of all data members of class type;
• doesn’t do anything special for fundamental types.

```cpp
class Widget
{
    public:
    // ...

    ~Widget()      // The compiler generated destructor destroys the
                   // string member, but doesn’t perform any special
                   // action for the integer and pointer ->
                   // potential resource leak!
    {
    }

    // ...;

    private:       // The three data members:
        int i;     // - i as a representative of a fundamental type
        std::string s;  // - s as a representative of a class (user-defined) type
        Resource* pr{}; // - pr as representative of a possible resource
};
```
The compiler generated destructor ...
• calls the destructor of all data members of class type;
• doesn’t do anything special for fundamental types.

```cpp
class Widget
{
public:

    // ...

    ~Widget() {       // The compiler generated destructor destroys the
        ~delete pr;   // string member, but doesn’t perform any special
        // action for the integer and pointer ->
        // potential resource leak!
    }

    // ...

private:          // The three data members:

    int i;         // - i as a representative of a fundamental type
    std::string s;  // - s as a representative of a class (user-defined) type
    Resource* pr{}; // - pr as representative of a possible resource
};
```
**Guideline:** Provide a manual destructor if there are any outstanding responsibilities that are not handled by the destructor of a class type data member.

**Guideline:** Never provide an empty destructor. Prefer to let the compiler provide a definition or define by `=default`. 
The Copy Ctor and Copy Assignment Operator
The Signatures of the Copy Operations

The signature of the copy constructor:

\[ \text{Widget} \ (\text{Widget} \ \text{const}& \ ); \quad \text{// The default} \]

\[ \text{Widget} \ (\text{Widget}& \ ); \quad \text{// Possible, but very likely not} \]
\[ \quad \text{// reasonable} \]

\[ \text{Widget} \ (\text{Widget} \ ); \quad \text{// Not possible; recursive call} \]

The signature of the copy assignment operator:

\[ \text{Widget}& \ \text{operator}=\ (\text{Widget} \ \text{const}& \ ); \quad \text{// The default} \]

\[ \text{Widget}& \ \text{operator}=\ (\text{Widget}& \ ); \quad \text{// Possible, but very likely not} \]
\[ \quad \text{// reasonable} \]

\[ \text{Widget}& \ \text{operator}=\ (\text{Widget} \ ); \quad \text{// Reasonable; builds on the} \]
\[ \quad \text{// copy constructor} \]
The Copy Ctor and Copy Assignment Operator

The compiler always generates the copy operations ...

// Compiler-generated copy ctor and copy assignment available
class Widget
{
  public:

    // ...

};

Widget w1{};
Widget w2( w1 ); // Compiler generated, ok
w1 = w2;        // Compiler generated, ok
The Copy Ctor and Copy Assignment Operator

The compiler \textbf{always} generates the copy operations ...

- if they are not explicitly declared and ...
- if no move operation is declared and ...
- if all bases/data members can be copy constructed/assigned.

// Compiler-generated copy ctor and copy assignment \textbf{not} available
class Widget
{
  public:
    Widget( Widget const& );
    Widget& operator=( Widget const& );
    // ... 
};

Widget w1{};
Widget w2( w1 ); // Explicitly defined, ok
w1 = w2; // Explicitly defined, ok
The Copy Ctor and Copy Assignment Operator

The compiler always generates the copy operations ...
• if they are not explicitly declared and ...
• if no move operation is declared and ...
• if all bases/data members can be copy constructed/assigned.

```cpp
// Compiler-generated copy ctor and copy assignment not available
class Widget
{
public:
    // Widget( Widget const& ) = delete;
    // Widget& operator=( Widget const& ) = delete;
    // ...
    Widget( Widget&& w ) noexcept;
};

Widget w1{};
Widget w2( w1 );  // Compiler error: Copy constructor not available
w1 = w2;          // Compiler error: Copy assignment not available
```
The Copy Ctor and Copy Assignment Operator

The compiler always generates the copy operations ...
• if they are not explicitly declared and ...
• if no move operation is declared and ...
• if all bases/data members can be copy constructed/assigned.

// Compiler-generated copy ctor and copy assignment not available
class Widget
{
  public:
    // Widget( Widget const& ) = delete;
    // Widget& operator=( Widget const& ) = delete;
  private:
    NonCopyable member_; // Data member without copy operations
};

Widget w1{};
Widget w2( w1 ); // Compiler error: Copy constructor not available
w1 = w2;         // Compiler error: Copy assignment not available
The Copy Ctor and Copy Assignment Operator

The compiler **always** generates the copy operations ...

- if they are not explicitly declared and ...
- if no move operation is declared and ...
- if all bases/data members can be copy constructed/assigned.

```cpp
// Compiler-generated copy ctor and copy assignment **not** available
class Widget
{
  public:
    // Widget( Widget const& ) = delete;
    // Widget& operator=( Widget const& ) = delete;
  private:
    NonCopyable member_; // Data member without copy operations
};
```

**Guideline:** Every class has a copy constructor and a copy assignment operator. Either they are available or (implicitly) deleted.
The Default Implementation

class Widget : public Base
{
  public:
  Widget( Widget const& other )
  : Base( other )
  , i{ other.i }
  , s{ other.s }
  , pr{ other.pr }
  {}

  Widget& operator=( Widget const& other )
  {
    Base::operator=( other );
    i = other.i;
    s = other.s;
    pr = other.pr;
    return *this;
  }

  // ...

private:
  // The three data members:
  int i;    // - i as a representative of a fundamental type
  std::string s; // - s as a representative of a class (user-defined) type
  Resource* pr{}; // - pr as representative of a possible resource
};
The Manual Implementation

class Widget : public Base
{
public:
    Widget( Widget const& other )
    : Base{ other }
    , i{ other.i }               // The default copy constructor performs
    , s{ other.s }               // a member-wise copy construction of
    , pr{ other.pr }             // all data members

    Widget& operator=( Widget const& other )
    {
        Base::operator=( other );
        i = other.i;            // The default copy assignment operator
        s = other.s;            // performs a member-wise copy assignment
        pr = other.pr;          // of all data members
        return *this;
    }

    ~Widget() { delete pr; }

    // ...

private:                // The three data members:
    int i;                 // - i as a representative of a fundamental type
    std::string s;        // - s as a representative of a class (user-defined) type
    Resource* pr{};       // - pr as representative of a possible resource
};
The Manual Implementation

class Widget : public Base
{
    public:
    Widget( Widget const& other )
        : Base{ other }
        , i { other.i }
        , s { other.s }
        , pr{ other.pr }
    {}

    Widget& operator=( Widget const& other )
    {
        Base::operator=( other );
        i = other.i;            // Shallow copy!!!
        s = other.s;            // This might result in
        pr = other.pr;          // a double delete!
        return *this;
    }

    ~Widget() { delete pr; }

    // ...
    private:                   // The three data members:
        int i;                 // - i as a representative of a fundamental type
        std::string s;        // - s as a representative of a class (user-defined) type
        Resource* pr{};       // - pr as representative of a possible resource
};
The Manual Implementation

```cpp
class Widget : public Base
{
public:
    Widget( Widget const& other )
    : Base{ other }
    , i { other.i }
    , s { other.s }
    , pr{ other.pr ? new Resource{*other.pr} : nullptr }
    {};
Widget& operator=( Widget const& other )
{
    Base::operator=( other );
    i = other.i;
    s = other.s;
    pr = ( other.pr ? new Resource{*other.pr} : nullptr );
    return *this;
}
~Widget() { delete pr; }

// ...
private: // The three data members:
    int i; // - i as a representative of a fundamental type
    std::string s; // - s as a representative of a class (user-defined) type
    Resource* pr{}; // - pr as representative of a possible resource
};
```

// The three data members:
// - i as a representative of a fundamental type
// - s as a representative of a class (user-defined) type
// - pr as representative of a possible resource
class Widget : public Base
{
public:
    Widget( Widget const& other )
    : Base{ other }
    , i { other.i }
    , s { other.s }
    , pr{ other.pr ? new Resource{*other.pr} : nullptr }
{
}
    Widget& operator=( Widget const& other )
    {
        delete pr; // Suspicious: Cleanup outside a destructor!
        Base::operator=( other ); // This now results in a problem for self-assignment!
        i = other.i; // Typical self-assignment check with if(this!=&other)
        s = other.s;
        pr = ( other.pr ? new Resource{*other.pr} : nullptr );
        return *this;
    }
    ~Widget() { delete pr; }

    // ...
private: // The three data members:
    int i; // - i as a representative of a fundamental type
    std::string s; // - s as a representative of a class (user-defined) type
    Resource* pr{}; // - pr as representative of a possible resource
};
The Manual Implementation

class Widget : public Base
{
public:
    Widget( Widget const& other )
        : Base{ other }
        , i{ other.i }
        , s{ other.s }
        , pr{ other.pr ? new Resource{*other.pr} : nullptr }
    {}
    Widget& operator=( Widget const& other )
    {
        // Temporary-swap idiom
        Widget tmp( other );  // This is not the fastest possible solution!
        swap( tmp );

        return *this;
    }

    ~Widget() { delete pr; }

    // ...
private:  // The three data members:
    int i;   // - i as a representative of a fundamental type
    std::string s;  // - s as a representative of a class (user-defined) type
    Resource* pr{};  // - pr as representative of a possible resource
};
class Widget : public Base
{
public:
    Widget( Widget const& other )
        : Base{ other }
            , i { other.i }
            , s { other.s }
            , pr{ other.pr ? new Resource{*other.pr} : nullptr }
    {}

    Widget& operator=( Widget const& other )
    {
        if( pr && other.pr ) {
            Base::operator=( other );  // No need to handle self-assignment explicitly
            i = other.i;  // if all bases and data members can handle self
            s = other.s;  // assignment on their own
            *pr = *other.pr;  // Copy assignment of the resources
        } else {
            // Temporary-swap idiom
            Widget tmp( other );
            swap( tmp );
        }
        return *this;
    }

    ~Widget();
};
The Manual Implementation

class Widget : public Base
{
public:
    Widget( Widget const& other );
    Widget& operator=( Widget const& other );
    ~Widget();
    void swap( Widget& other )
    {
        using std::swap;
        swap( id, other.id );
        swap( name, other.name );
        swap( resource, other.resource );
    }
};

Core Guideline C.83: For value-like types, consider providing a noexcept swap function

// ... private:
    // The three data members:
    int i;    // - i as a representative of a fundamental type
    std::string s;    // - s as a representative of a class (user-defined) type
    Resource* pr{};    // - pr as representative of a possible resource
};
The Manual Implementation

```cpp
class Widget : public Base {
public:
  Widget( Widget const& other );
  Widget& operator=( Widget const& other );
  ~Widget();

private:
  int i;
  std::string s;
  Resource* pr{ }; // - pr as representative of a possible resource
};
```

Guideline: Take care of the Rule of 3: When you require a destructor, you most probably also require a copy constructor and copy assignment operator.

Outdated since C++11?
class Widget : public Base
{
public:
    Widget( Widget const& other );
    Widget& operator=( Widget const& other );
~Widget();

private:
    int i;
    std::string s;
    std::unique_ptr<Resource> pr{ };
class Widget : public Base
{
    public:
    Widget( Widget const& other);
    Widget& operator=( Widget const& other);

    // ... 
    private:
    int i;
    std::string s;
    std::shared_ptr<Resource> pr{];
};

// Note that this fundamentally changes
// the semantics of the class
The Manual Implementation

class Widget : public Base
{
    public:

Guideline: Strive for the Rule of 0: Classes that don’t require an explicit destructor, copy constructor and copy assignment operator are much (!) easier to handle.

    // ...
    private:
    int i;
    std::string s;
    std::shared_ptr<Resource> pr{};
};
The Move Ctor and Move Assignment Operator
The Signatures of the Move Operations

The signature of the move constructor:

```cpp
Widget( Widget&& ) noexcept;       // The default
Widget( Widget const&& ) noexcept; // Possible, but uncommon
```

The signature of the move assignment operator:

```cpp
Widget& operator=( Widget&& ) noexcept;       // The default
Widget& operator=( Widget const&& ) noexcept; // Also uncommon
```
The Move Ctor and Move Assignment Operator

The compiler generates the move operations ...

```cpp
// Compiler-generated move ctor and move assignment available
class Widget
{
  public:

    // ...

};

Widget w1{};
Widget w2( std::move(w1) ); // Compiler generated, ok
w1 = std::move(w2); // Compiler generated, ok
```
The Move Ctor and Move Assignment Operator

The compiler generates the move operations ...
• if they are not explicitly declared and ...
• if no destructor and no copy operation is declared and ...
• if all bases/data members can be copy or move constructed/assigned.

```cpp
// Compiler-generated move ctor and move assignment available
class Widget
{
public:
    Widget( Widget&& ) noexcept;
    Widget& operator=( Widget&& ) noexcept;
    // ...
};

Widget w1{};
Widget w2( std::move(w1) );  // Explicitly defined, ok
w1 = std::move(w2);          // Explicitly defined, ok
```
The Move Ctor and Move Assignment Operator

The compiler generates the move operations ... 
• if they are not explicitly declared and ...
• if no destructor and no copy operation is declared and ...
• if all bases/data members can be copy or move constructed/assigned.

```cpp
// Compiler-generated move ctor and move assignment not available
class Widget {
public:
    Widget( Widget const& ); // or alternatively declaration of
    // ... // destructor or copy assignment

};

Widget w1{};
Widget w2( std::move(w1) ); // Copy ctor instead of move ctor
w1 = std::move(w2); // Copy assign instead of move assign
```
The Move Ctor and Move Assignment Operator

The compiler generates the move operations ...
• if they are not explicitly declared and ...
• if no destructor and no copy operation is declared and ...
• if all bases/data members can be copy or move constructed/assigned.

```cpp
// Compiler-generated copy ctor and copy assignment available
class Widget
{
public:
    // ...
private:
    NonCopyable member_; // Data member without copy operations
};

Widget w1{};
Widget w2( std::move(w1) ); // Compiler generated, ok
w1 = std::move(w2); // Compiler generated, ok
```
The Move Ctor and Move Assignment Operator

The compiler generates the move operations ...
• if they are not explicitly declared and ...
• if no destructor and no copy operation is declared and ...
• if all bases/data members can be copy or move constructed/assigned.

```cpp
// Compiler-generated copy ctor and copy assignment available
class Widget
{
public:
   // ...
private:
   NonMovable member_; // Data member without move operations
};

Widget w1{};
Widget w2( std::move(w1) );  // Copy ctor instead of move ctor
w1 = std::move(w2);  // Copy assign instead of move assign
```
The compiler generates the move operations ...
• if they are not explicitly declared and ...
• if no destructor and no copy operation is declared and ...
• if all bases/data members can be copy or move constructed/assigned.

// Compiler-generated copy ctor and copy assignment not available
class Widget
{
public:
    // ...
private:
    Immobile member_;  // Data member without copy AND move ops
};

Widget w1{};
Widget w2( std::move(w1) );  // Compiler error: No move ctor
w1 = std::move(w2);          // Compiler error: No move assignment
The Default Implementation

class Widget : public Base
{
public:
    Widget( Widget&& other ) noexcept
    : Base{ std::move(other) }
    , i { std::move(other.i) } // The default move constructor performs
    , s { std::move(other.s) } // a member-wise move construction of
    , pr{ std::move(other.pr) } // all data members

    Widget& operator=( Widget&& other ) noexcept
    {
        Base::operator=(std::move(other));
        i = std::move(other.i);       // The default move assignment operator
        s = std::move(other.s);       // performs a member-wise move assignment
        pr = std::move(other.pr);     // of all data members
        return *this;
    }

    // ...

private: // The three data members:
    int i;    // - i as a representative of a fundamental type
    std::string s; // - s as a representative of a class (user-defined) type
    Resource* pr{ }; // - pr as representative of a possible resource
};
The Default Implementation

class Widget : public Base
{
    public:
        Widget( Widget&& other ) noexcept
        : Base{ std::move(other) }
        , i { std::move(other.i) }
        , s { std::move(other.s) }
        , pr{ std::move(other.pr) }
    }
    Widget& operator=( Widget&& other ) noexcept
    {
        Base::operator=(std::move(other));
        i = std::move(other.i); // This might result in
        s = std::move(other.s); // a double delete!
        pr = std::move(other.pr);
        return *this;
    }
    ~Widget() { delete resource; }

    // ...
    private:
        int i; // - i as a representative of a fundamental type
        std::string s; // - s as a representative of a class (user-defined) type
        Resource* pr{ }; // - pr as representative of a possible resource
};
The Default Implementation

class Widget : public Base
{
public:
    Widget( Widget&& other ) noexcept
    : Base{ std::move(other) },
      i { std::move(other.i) },
      s { std::move(other.s) },
      pr{ std::exchange(other.pr,{}) }
    {}

    Widget& operator=( Widget&& other ) noexcept
    {
        delete pr; // Suspicious: Cleanup outside a destructor!
        Base::operator=(std::move(other));
        i = std::move(other.i);
        s = std::move(other.s);
        pr = std::exchange(other.pr,{}); // Problem in case of self-assignment!
        return *this;
    }

    ~Widget() { delete resource; }

    // ...

private: // The three data members:
    int i;  // - i as a representative of a fundamental type
    std::string s; // - s as a representative of a class (user-defined) type
    Resource* pr{}; // - pr as representative of a possible resource
};
The Default Implementation

class Widget : public Base
{
public:
    Widget( Widget&& other ) noexcept
        : Base{ std::move(other) }
        , i { std::move(other.i) }
        , s { std::move(other.s) }
        , pr{ std::exchange(other.pr,{}) }
    {}
Widget& operator=( Widget&& other ) noexcept
    {
        delete pr;
        Base::operator=(std::move(other));
        i = std::move(other.i);
        s = std::move(other.s);
        pr = other.pr; other.pr = nullptr;
        return *this;
    }
~Widget() { delete resource; }

    // ...
private:
    // The three data members:
    int i; // - i as a representative of a fundamental type
    std::string s; // - s as a representative of a class (user-defined) type
    Resource* pr{}; // - pr as representative of a possible resource
};
The Default Implementation

class Widget : public Base
{
public:
    Widget( Widget&& other ) noexcept
    : Base{ std::move(other) }
    , i{ std::move(other.i) }
    , s{ std::move(other.s) }
    , pr{ std::exchange(other.pr,{} ) }
    {}
Widget& operator=( Widget&& other ) noexcept
{
    // Temporary-swap idiom
    Widget tmp( std::move(other) );
    swap( tmp );

    return *this;
}

~Widget() { delete resource; }

    // ... 
private:    // The three data members:
    int i;    // - i as a representative of a fundamental type
    std::string s;    // - s as a representative of a class (user-defined) type
    Resource* pr{};    // - pr as representative of a possible resource
};
class Widget : public Base
{
  public:
    Widget( Widget&& other ) noexcept;
    Widget& operator=( Widget&& other ) noexcept;
    ~Widget();
    Widget( Widget const& other );
    Widget& operator=( Widget const& other );

  private: // The three data members:
    int i; // - i as a representative of a fundamental type
    std::string s; // - s as a representative of a class (user-defined) type
    Resource* pr{}; // - pr as representative of a possible resource
};
The Default Implementation

class Widget : public Base
{
public:
    Widget( Widget&& other ) noexcept;

    Widget& operator=( Widget&& other ) noexcept;

    ~Widget();

    Widget( Widget const& other );

    Widget& operator=( Widget const& other );

private:
    // The three data members:
    int i;       // - i as a representative of a fundamental type
    std::string s;  // - s as a representative of a class (user-defined) type
    Resource* pr{};  // - pr as representative of a possible resource
};

Guideline: Take care of the Rule of 5: When you require a destructor, you most probably also require the two copy operations and the two move operations.
The Default Implementation

class Widget : public Base
{
    public:
    Widget( Widget&& other ) noexcept;

    Widget& operator=( Widget&& other ) noexcept;

    ~Widget();

    Widget( Widget const& other );

    Widget& operator=( Widget const& other );

    Guideline: Take care of the Rule of 5: When you require a destructor, you most probably also require the two copy operations and the two move operations.

    // ...
    private:
    int i;
    std::string s;
    Resource* pr{};  // Manual resource handling is suspicious!
};
The Default Implementation

class Widget : public Base
{
  public:
    Widget( Widget&& other ) noexcept;
    Widget& operator=( Widget&& other ) noexcept;
    ~Widget();
    Widget( Widget const& other );
    Widget& operator=( Widget const& other );

  // ...
  private:
    int i;
    std::string s;
    std::unique_ptr<Resource> pr{ };
The Default Implementation

class Widget : public Base
{
    public:

    Widget( Widget const& other );

    Widget& operator=( Widget const& other );

    // ... private:
    int i;
    std::string s;
    std::shared_ptr<Resource> pr{ }; 
};

// Note that this fundamentally changes
// the semantics of the class
Guideline: Strive for the **Rule of 0**: Classes that don’t require an explicit destructor, explicit copy operations and explicit move operations are much (!) easier to handle.

Guideline: Try to reduce the use of pointers!

Core Guideline R.21: Prefer `unique_ptr` over `shared_ptr` unless you need to share ownership
Guidelines
Core Guideline C.20: If you can avoid defining default operations, do so.

```cpp
template< typename T >
class Widget
{
    public:
        Widget( size_t size )
            : values_{new T[size]} , size_{size}
        {}

    ~Widget() { delete[] values_; }
    // ...

    private:
        T* values_;  // Manual resource handling is suspicious!
        size_t size_; }
};
```
Core Guideline C.20: If you can avoid defining default operations, do

```cpp
template< typename T >
class Widget
{
    public:
        Widget( size_t size )
            : values_{size}
{
        }

    // ...

    private:
        std::vector<T> values_; 

};
```
Core Guideline C.21: If you define or \texttt{=delete} any default operation, define or \texttt{=delete} them all

```cpp
template< typename T >
class Widget
{
    public:

        // Which special member functions do we need?

        // ...

    private:
        std::unique_ptr<T> value_;  // ...

};
```

The Rule of 5
Guidelines

Core Guideline C.21: If you define or delete any default operation, define or delete them all

```cpp
template< typename T >
class Widget
{
  public:
    Widget( Widget const& );
    Widget& operator=( Widget const& );

    // ... but the copy operations disable the move operations

  // ...

  private:
    std::unique_ptr<T> value_;  // ...
};
```

The Rule of 5
Guidelines

Core Guideline C.21: If you define or =delete any default operation, define or =delete them all

```
template< typename T >
class Widget
{
  public:
    Widget( Widget const& );
    Widget& operator=( Widget const& );
    Widget( Widget&& ) noexcept = default;
    Widget& operator=( Widget&& ) noexcept = default;
    ~Widget() = default;
    // ...

  private:
    std::unique_ptr<T> value_; // Note that =default defines
    // the special member function
    // ...
};
```

The Rule of 5
Core Guideline C.21: If you define or =delete any default operation, define or =delete them all

```cpp
#include <memory>

template< typename T >
class Widget
{
public:
  Widget( Widget const& ) = delete;
  Widget& operator=( Widget const& ) = delete;
  Widget( Widget&& ) noexcept = default;
  Widget& operator=( Widget&& ) noexcept = default;
  ~Widget() = default;
  // ...

private:
  std::unique_ptr<T> value_;  // ...
};
```
Guidelines

Note that it makes a difference whether you don’t provide or explicitly delete the move operations:

- **Move operations not provided:** When an object is moved, copy serves as a fallback
- **Move operations deleted:** Moving an object results in a compilation error

```cpp
template< typename T >
class Widget
{
    public:
        Widget( Widget const& ) = default;
        Widget& operator=( Widget const& ) = default;
        Widget( Widget&& ) = delete;
        Widget& operator=( Widget&& ) = delete;
        ~Widget() = default;
        // ...
};
```
Guidelines

**Guideline:** Follow the **Rule of 5** if you want to default/delete the move operations.

```cpp
template<typename T>
class Widget {
    public:
        Widget( Widget const& ) = default;
        Widget& operator=( Widget const& ) = default;
        Widget( Widget&& ) = delete;
        Widget& operator=( Widget&& ) = delete;
        ~Widget() = default;
        // ...
};
```
**Guidelines**

**Guideline:** Follow the **Rule of 5** if you want to default/delete the move operations.

**Guideline:** Follow the **Rule of 3** if you want to copy instead of move.

```cpp
template< typename T >
class Widget
{
    public:
    Widget( Widget const& ) = default;
    Widget& operator=( Widget const& ) = default;
    // Move constructor explicitly omitted
    // Move assignment operator explicitly omitted
    ~Widget() = default;
    // ...
};
```
Guidelines

**Guideline:** Be suspicious of manual resource cleanup (RAII)!

**Guideline:** Try to reduce the use of pointers!
Back to Basics: The Special Member Functions

KLAUS IGLBERGER